Concurrency

Dao Anh Tu
(tu.dao@lge.com)

2021. 11 . 23

VS DCV

AGENDA

I. Overview

II. Issue

III. Thread API

IV. Locks

V. Locked data structures

VI. Application

I - Overview

Why use multi-thread:

- Parallelism - improve performance.
- Avoid stuck.

Note: Threads share an address space and thus make it easy to share data, and hence are a natural choice when constructing these types of programs. Processes are a more sound choice for logically separate tasks where little sharing of data structures in memory is needed.

Difference Single-Threaded And Multi-Threaded: there is a number stack

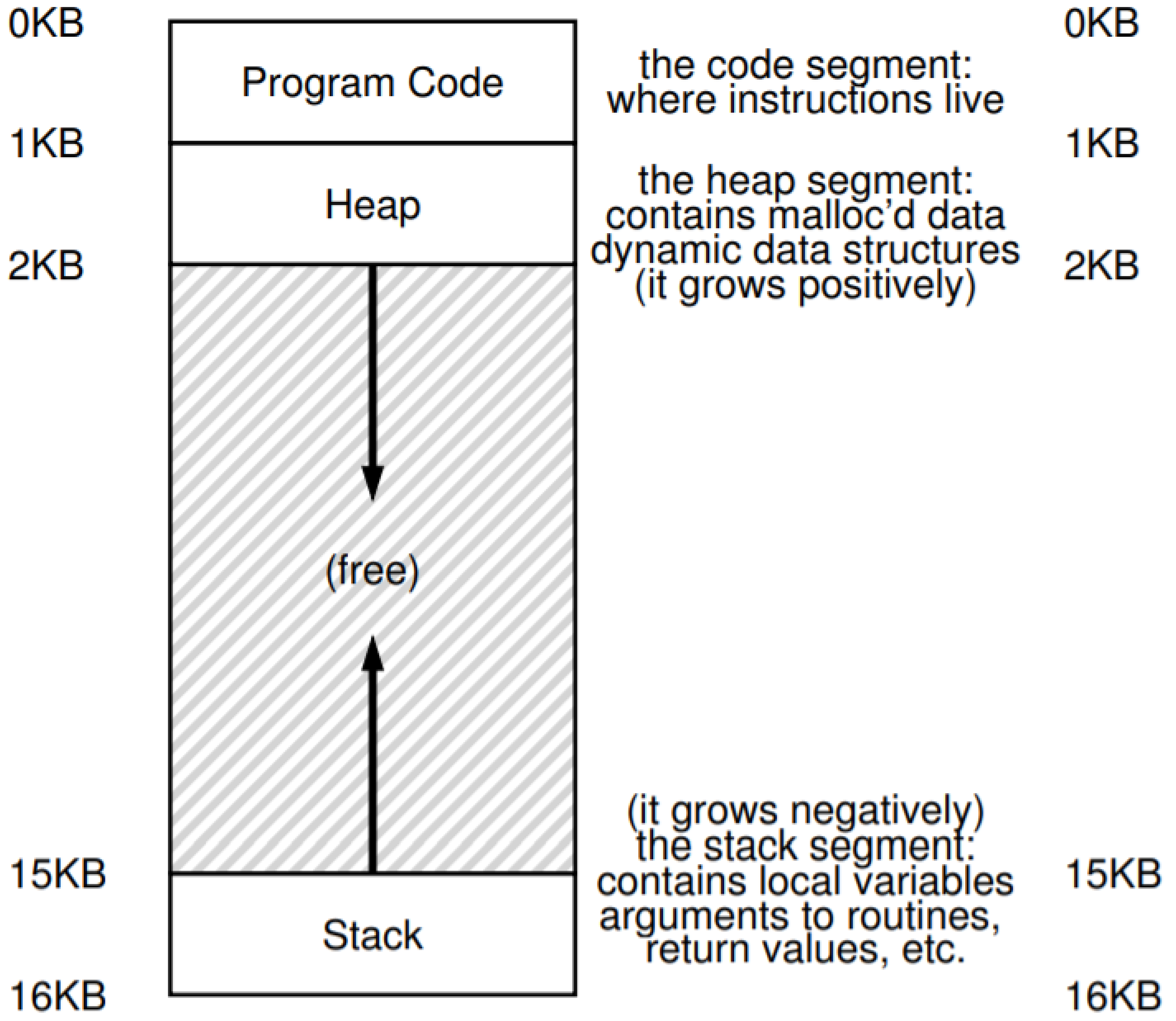
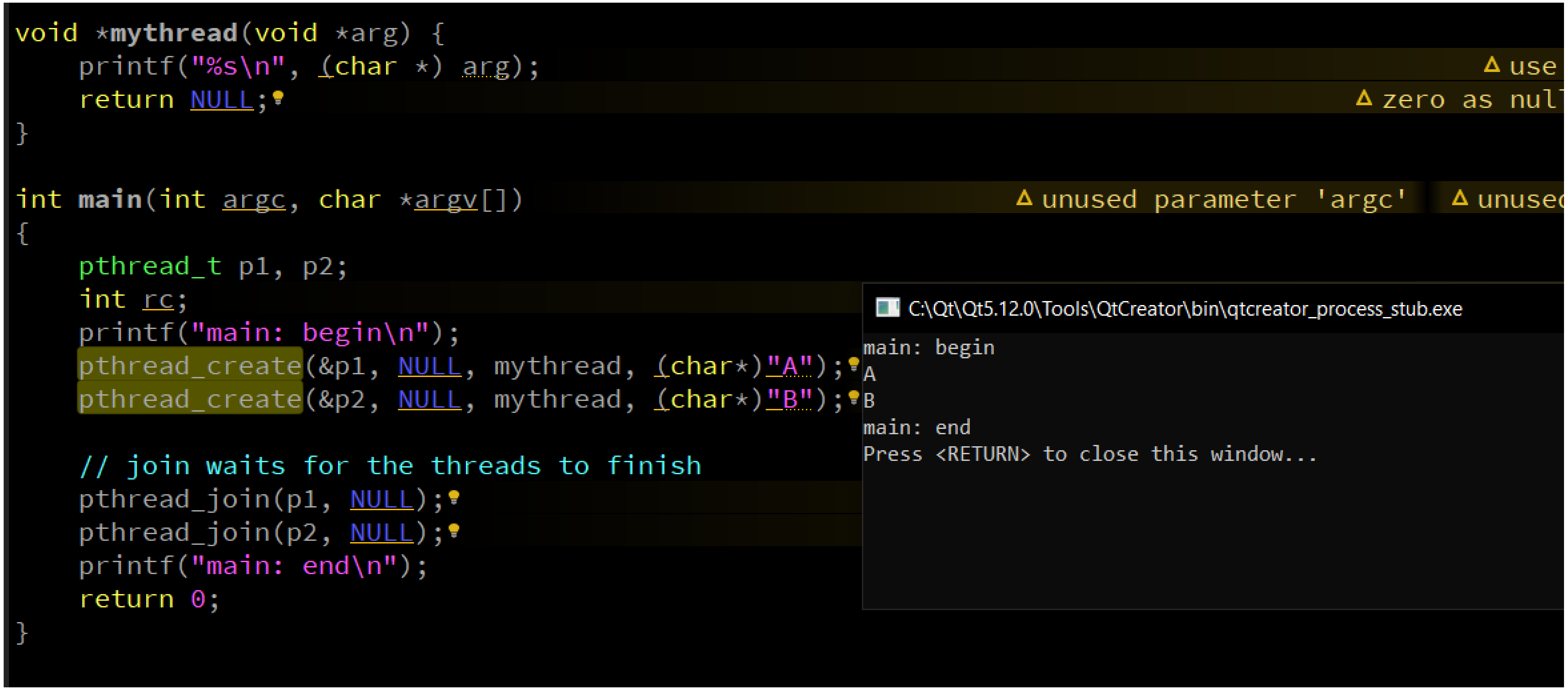

0KB
Program Code
the code segment: where instructions live
1KB
Heap
the heap segment: contains malloc'd data dynamic data structures (it grows positively)
2KB
(free)
(it grows negatively) the stack segment: contains local variables arguments to routines, return values, etc.
15KB
Stack
16KB
0KB
Program Code
1KB
Heap
2KB
(free)
Stack (2)
(free)
15KB
Stack (1)
16KB

Figure 26.1: **Single-Threaded And Multi-Threaded Address Spaces**

I - Overview

Code demo



```
void *mythread(void *arg) {
    printf("%s\n", (char *) arg);                                          ⚠ use
    return NULL;                                                        ⚠ zero as nul
}

int main(int argc, char *argv[])                  ⚠ unused parameter 'argc'  ⚠ unused
{
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    pthread_create(&p1, NULL, mythread, (char*)"A");
    pthread_create(&p2, NULL, mythread, (char*)"B");

    // join waits for the threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: end\n");
    return 0;
}
```

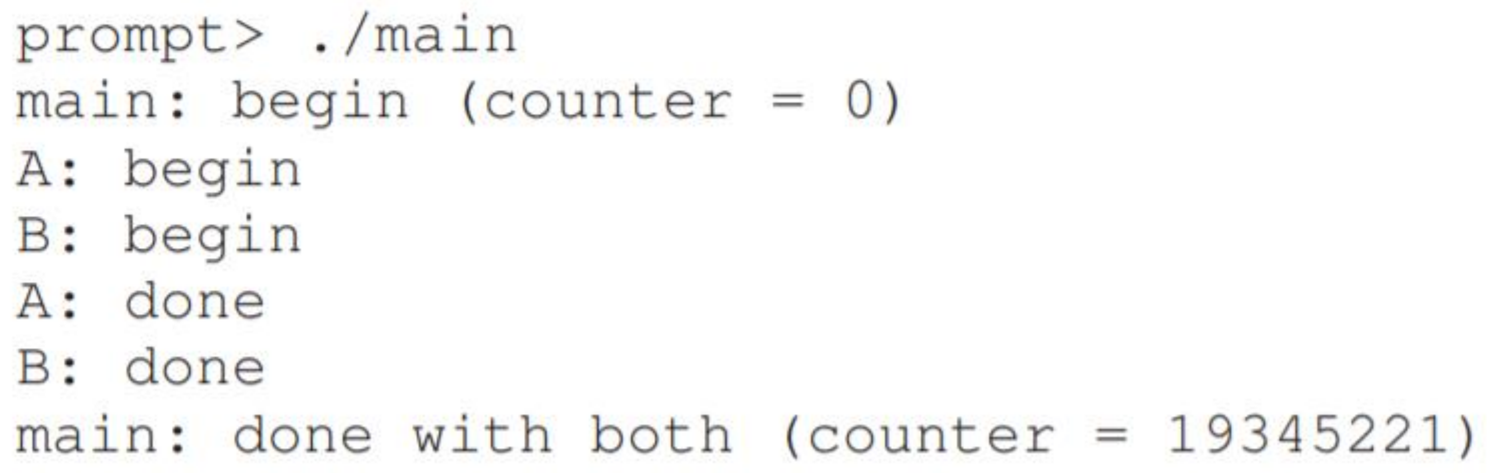
C:\Qt\Qt5.12.0\Tools\QtCreator\bin\qtcreator_process_stub.exe

```
main: begin
A
B
main: end
Press <RETURN> to close this window...
```

II - Issue

- **Memory shared:** What the happen when two threads access to one global variable?

Use 2 thread add a number to the shared variable counter, and do so 10 million times (1e7) in a loop. Thus, the desired final result is: 20,000,000. Unfortunately, we don’t necessarily get the desired result.



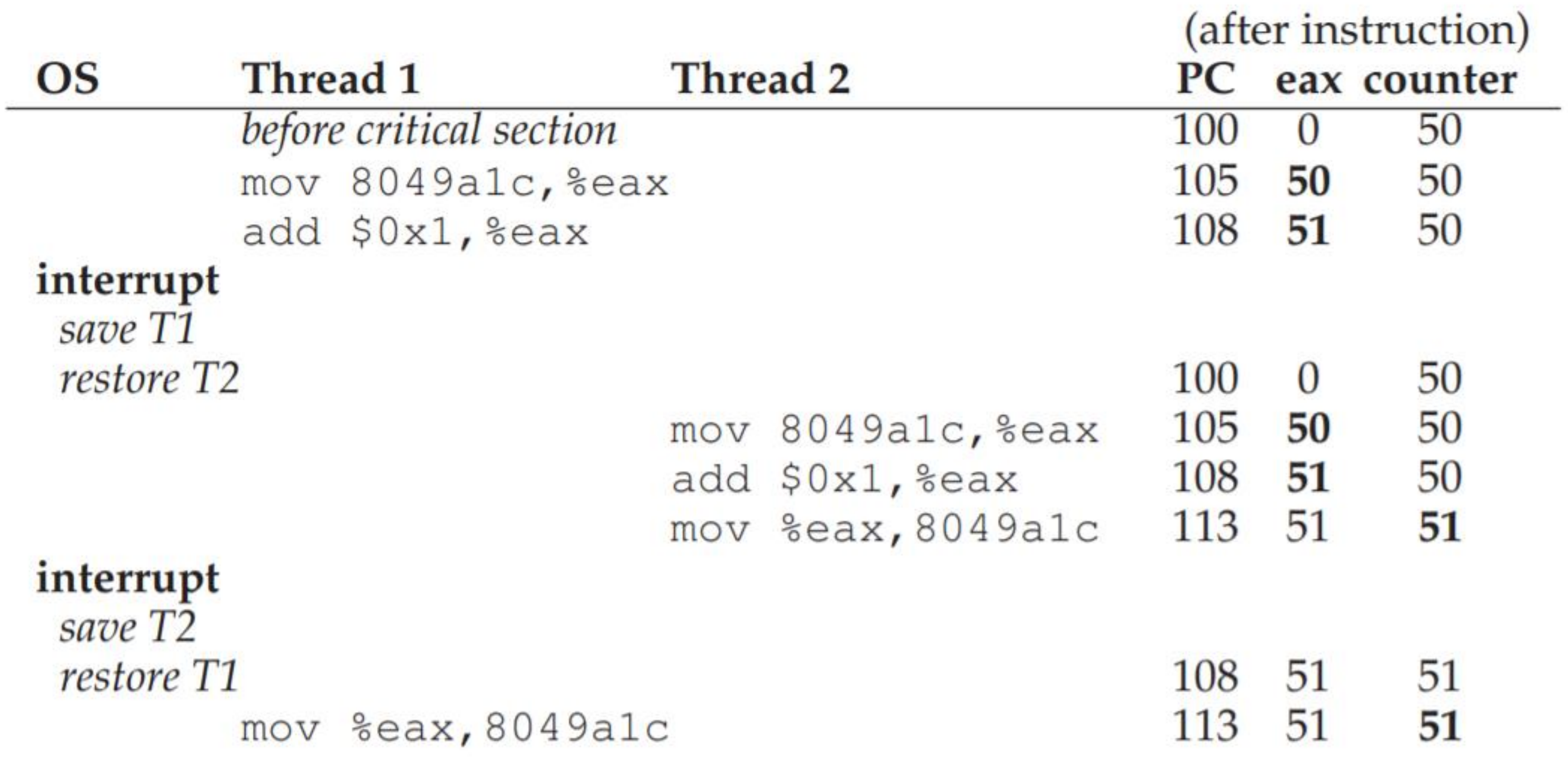
```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

II - Issue

The Heart Of The Problem: Uncontrolled Scheduling

The code sequence for doing

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```



			(after instruction)		
OS	**Thread 1**	**Thread 2**	**PC**	**eax**	**counter**
	before critical section		100	0	50
	`mov 8049a1c,%eax`		105	**50**	50
	`add $0x1,%eax`		108	**51**	50
interrupt					
save T1					
restore T2			100	0	50
		`mov 8049a1c,%eax`	105	**50**	50
		`add $0x1,%eax`	108	**51**	50
		`mov %eax,8049a1c`	113	51	**51**
interrupt					
save T2					
restore T1			108	51	51
	`mov %eax,8049a1c`		113	51	**51**

Figure 26.7: **The Problem: Up Close and Personal**

II - Issue

- **Data race:** multiple threads of execution enter the critical section at roughly the same time

Multiple threads executing this code can result in a race condition, we call this code a critical section. A critical section is a piece of code that accesses a shared variable (or more generally, a shared resource) and must not be concurrently executed by more than one thread.

- **Waiting For Another:** one thread must wait for another to complete some action before it continues.

When a process performs a disk I/O and is put to sleep; when the I/O completes, the process needs to be roused from its slumber so it can continue

II – Thread API

Pthread: some api helpfull

- **pthread_create()** - create new threads

```
#include <pthread.h>
int
pthread_create(pthread_t           *thread,
         const pthread_attr_t *attr,
               void           *(*start_routine)(void*),
               void           *arg);
```

- **pthread join() -** wait for a thread to complete

```
int pthread_join(pthread_t thread, void **value_ptr);
```

II – Thread API

- **pthread_mutex_lock()**
- **pthread_mutex_unlock()**

Mutual exclusion to a critical section via locks

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- **pthread_cond_wait()**
- **pthread_cond_signal()**

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

Condition variables are useful when some kind of signaling must take place between threads, if one thread is waiting for another to do something before it can continue.

III – THREAT API

Thread function

```
void foo(param)
{
		// Do something
}

// The parameters to the function are put after the comma
std::thread thread_obj(foo, params);
```

```
std::thread thread_object([](params) {
		// Do Something
};, params);
```

III – THREAT API

Thread function of class

```
// Define the class of function object
class fn_object_class {
            // Overload () operator
            void operator()(params)
            {
                        // Do Something
            }
}

// Create thread object
std::thread thread_object(fn_object_class(), params)
```

IV - LOCK

Concept

A lock is just a **variable** holds the state of the lock at any instant in time. It is either **unlocked** and thus no thread holds the lock, or **locked**, and thus exactly one thread holds the lock and presumably is in a critical section

```
lock_t mutex; // some globally-allocated lock ’mutex’
…
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

IV - LOCK

Evaluating Locks

Basic criteria to evaluate the lock:

- **Mutual exclusion:** does the lock work, preventing multiple threads from entering a critical section?
- **Fairness:** does any thread contending for the lock starve while doing so, thus never obtaining it?
- **Performance:** specifically the time overheads added by using the lock

IV - LOCK

Build a lock

- **Controlling Interrupts:** the solutions used to provide mutual exclusion was to disable interrupts for critical sections

```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

The negatives:

- This approach requires us to allow any calling thread to perform a privileged operation.
- The approach does not work on multiprocessors
- Turning off interrupts for extended periods of time can lead to interrupts becoming lost, which can lead to serious systems problems

IV - LOCK

Build a lock

- **Using Loads/Stores:** use a simple variable (flag) to indicate whether some thread has possession of a lock

```
void lock(lock_t *mutex) {
    while (mutex->flag == 1)   // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;           // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

Figure 28.1: **First Attempt: A Simple Flag**

The negatives:

➢The **correctness** problem.

➢The **performance** problem

IV - LOCK

Build a lock

- **Building Working Spin Locks:** returns the old value pointed to by the old ptr, and simultaneously updates said value to new.

```
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new;     // store 'new' into old_ptr
    return old;         // return the old value
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Figure 28.3: **A Simple Spin Lock Using Test-and-set**

IV - LOCK

Build the spin Locks

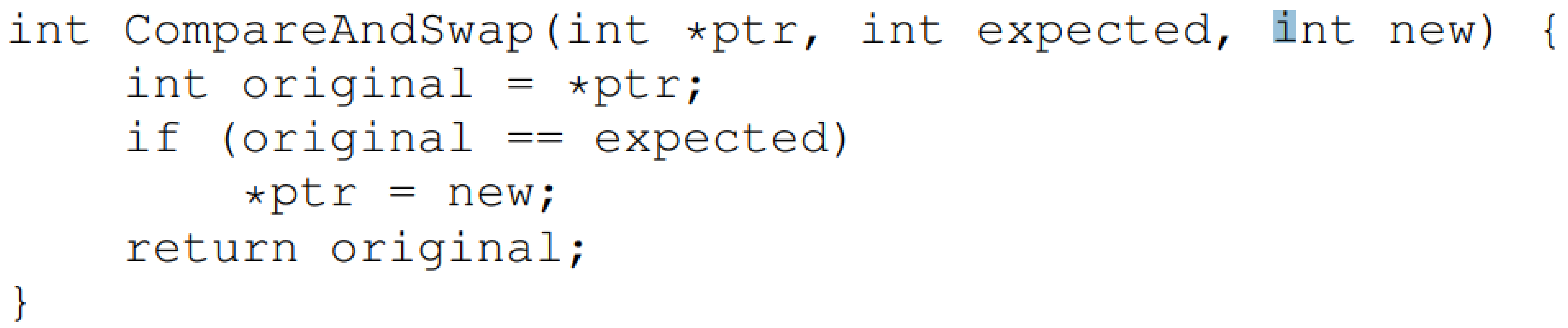
Basic criteria to evaluate the spin locks:

- **Correctness:** the spin lock only allows a single thread to enter the critical section at a time. Thus, we have a correct lock.
- **Fairness:** a thread spinning may spin forever, under contention, another lock may to starvation.
- **Performance:** in the single CPU case, performance overheads can be quite painful; on multiple CPUs, spin locks work reasonably well.

IV - LOCK

Build the spin Locks

- **Compare-And-Swap:** test whether the value at the address specified by ptr is equal to expected.



```
int CompareAndSwap(int *ptr, int expected, int new) {
    int original = *ptr;
    if (original == expected)
        *ptr = new;
    return original;
}
```

Figure 28.4: **Compare-and-swap**

IV - LOCK

Build the spin Locks

- **Load-Linked and Store-Conditional:**

Load-Linked much like a typical load instruction, and simply fetches a value from memory and places it in a register

Store-Conditional which only succeeds (and updates the value stored at the address just load-linked from), if no intervening store to the address has taken place

```
int LoadLinked(int *ptr) {
    return *ptr;
}

int StoreConditional(int *ptr, int value) {
    if (no update to *ptr since LoadLinked to this address) {
        *ptr = value;
        return 1; // success!
    } else {
        return 0; // failed to update
    }
}
```

Figure 28.5: **Load-linked And Store-conditional**

IV - LOCK

Build the spin Locks

- **Load-Linked and Store-Conditional:**

```
void lock(lock_t *lock) {
    while (1) {
        while (LoadLinked(&lock->flag) == 1)
            ; // spin until it's zero
        if (StoreConditional(&lock->flag, 1) == 1)
            return; // if set-it-to-1 was a success: all done
                    // otherwise: try it all over again
    }
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Figure 28.6: **Using LL/SC To Build A Lock**

IV - LOCK

Build the spin Locks

- **Fetch-And-Add:** atomically increments a value while returning the old value at a particular address.

```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

IV - LOCK

Too Much Spinning:

- **Just yield:** we assume an operating system primitive yield() which a thread can call when it wants to give up the CPU and let another thread run

```
void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}

void unlock() {
    flag = 0;
}
```

Figure 28.8: **Lock With Test-and-set And Yield**

IV - LOCK

Using Queues: Sleeping Instead Of Spinning

```
void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}

void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (queue_empty(m->q))
        m->flag = 0; // let go of lock; no one wants it
    else
        unpark(queue_remove(m->q)); // hold lock
                                    // (for next thread!)
    m->guard = 0;
}
```

Figure 28.9: **Lock With Queues, Test-and-set, Yield, And Wakeup**

IV - LOCK

Using Queues: Sleeping Instead Of Spinning

- **Negatives** : **wakeup/waiting race**.
- Solves: add setpark(): a thread can indicate it is about to park. If it then happens to be interrupted and another thread calls unpark before park is actually called, the subsequent park returns immediately instead of sleeping.

```
queue_add(m->q, gettid());
setpark(); // new code
m->guard = 0;
```

V - Locked data structures

- **Concurrent Counters:** When a thread running on a given core wishes to increment the counter, it increments its local counter; access to this local counter is synchronized via the corresponding local lock.

```
// update: usually, just grab local lock and update
// local amount; once local count has risen 'threshold',
// grab global lock and transfer local values to it
void update(counter_t *c, int threadID, int amt) {
    int cpu = threadID % NUMCPUS;
    pthread_mutex_lock(&c->llock[cpu]);
    c->local[cpu] += amt;
    if (c->local[cpu] >= c->threshold) {
        // transfer to global (assumes amt>0)
        pthread_mutex_lock(&c->glock);
        c->global += c->local[cpu];
        pthread_mutex_unlock(&c->glock);
        c->local[cpu] = 0;
    }
    pthread_mutex_unlock(&c->llock[cpu]);
}

// get: just return global amount (approximate)
int get(counter_t *c) {
    pthread_mutex_lock(&c->glock);
    int val = c->global;
    pthread_mutex_unlock(&c->glock);
    return val; // only approximate!
}
```

Figure 29.4: **Approximate Counter Implementation**

V - Locked data structures

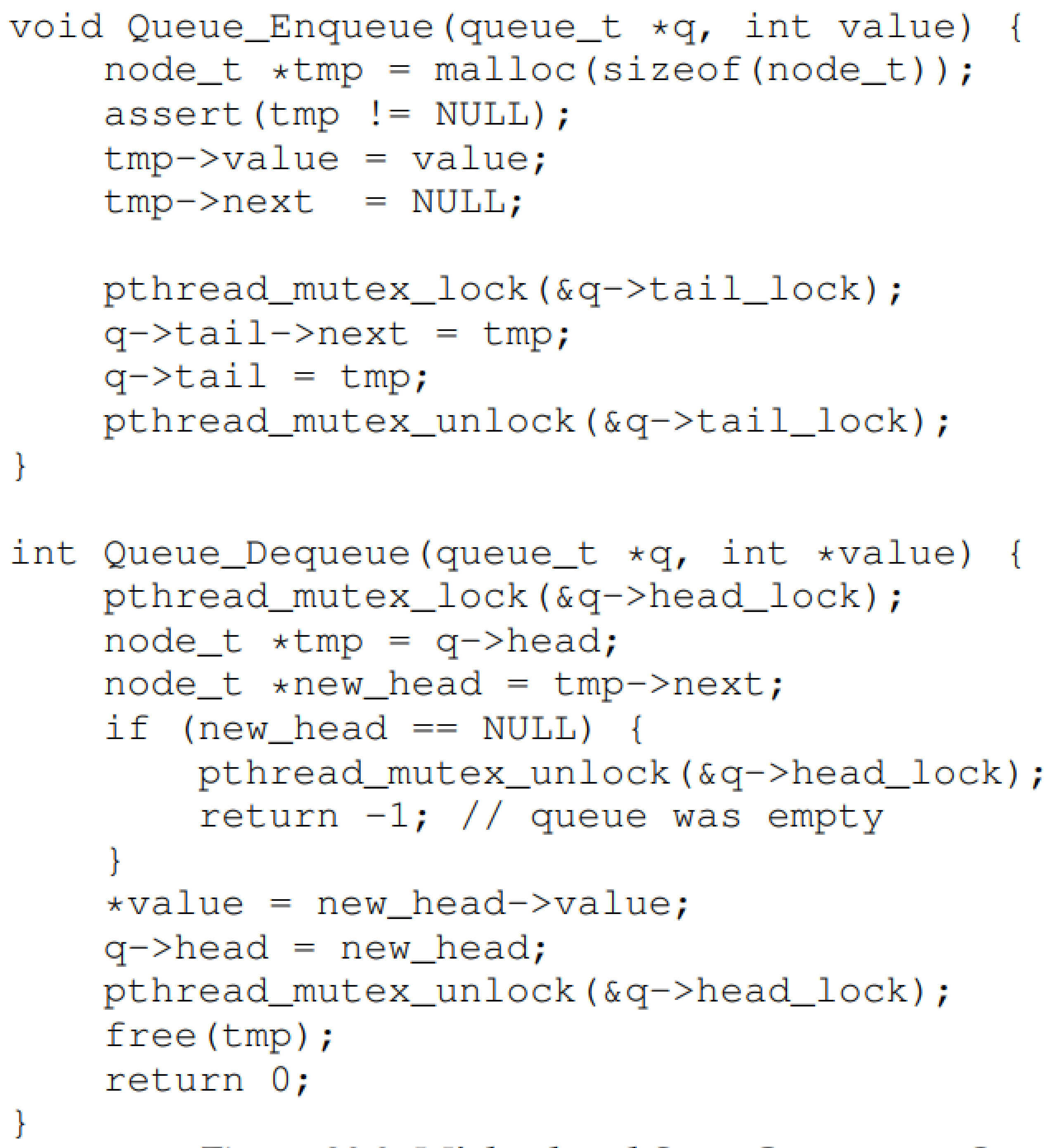
- **Concurrent Linked Lists:** acquires a lock in the insert routine upon entry, and releases it upon exit. One small tricky issue arises if malloc() happens to fail (a rare case); in this case, the code must also release the lock before failing the insert.

```
void List_Insert(list_t *L, int key) {
    // synchronization not needed
    node_t *new = malloc(sizeof(node_t));
    if (new == NULL) {
        perror("malloc");
        return;
    }
    new->key = key;

    // just lock critical section
    pthread_mutex_lock(&L->lock);
    new->next = L->head;
    L->head   = new;
    pthread_mutex_unlock(&L->lock);
}

int List_Lookup(list_t *L, int key) {
    int rv = -1;
    pthread_mutex_lock(&L->lock);
    node_t *curr = L->head;
    while (curr) {
        if (curr->key == key) {
            rv = 0;
            break;
        }
        curr = curr->next;
    }
    pthread_mutex_unlock(&L->lock);
    return rv; // now both success and failure
}
```

Figure 29.8: **Concurrent Linked List: Rewritten**

V - Locked data structures

- **Concurrent Queues:** there are two locks, one for the head of the queue, and one for the tail. The goal of these two locks is to enable concurrency of enqueue and dequeue operations.



```
void Queue_Enqueue(queue_t *q, int value) {
    node_t *tmp = malloc(sizeof(node_t));
    assert(tmp != NULL);
    tmp->value = value;
    tmp->next  = NULL;

    pthread_mutex_lock(&q->tail_lock);
    q->tail->next = tmp;
    q->tail = tmp;
    pthread_mutex_unlock(&q->tail_lock);
}

int Queue_Dequeue(queue_t *q, int *value) {
    pthread_mutex_lock(&q->head_lock);
    node_t *tmp = q->head;
    node_t *new_head = tmp->next;
    if (new_head == NULL) {
        pthread_mutex_unlock(&q->head_lock);
        return -1; // queue was empty
    }
    *value = new_head->value;
    q->head = new_head;
    pthread_mutex_unlock(&q->head_lock);
    free(tmp);
    return 0;
}
```

Figure 29.9: **Michael and Scott Concurrent Queue**

V - Locked data structures

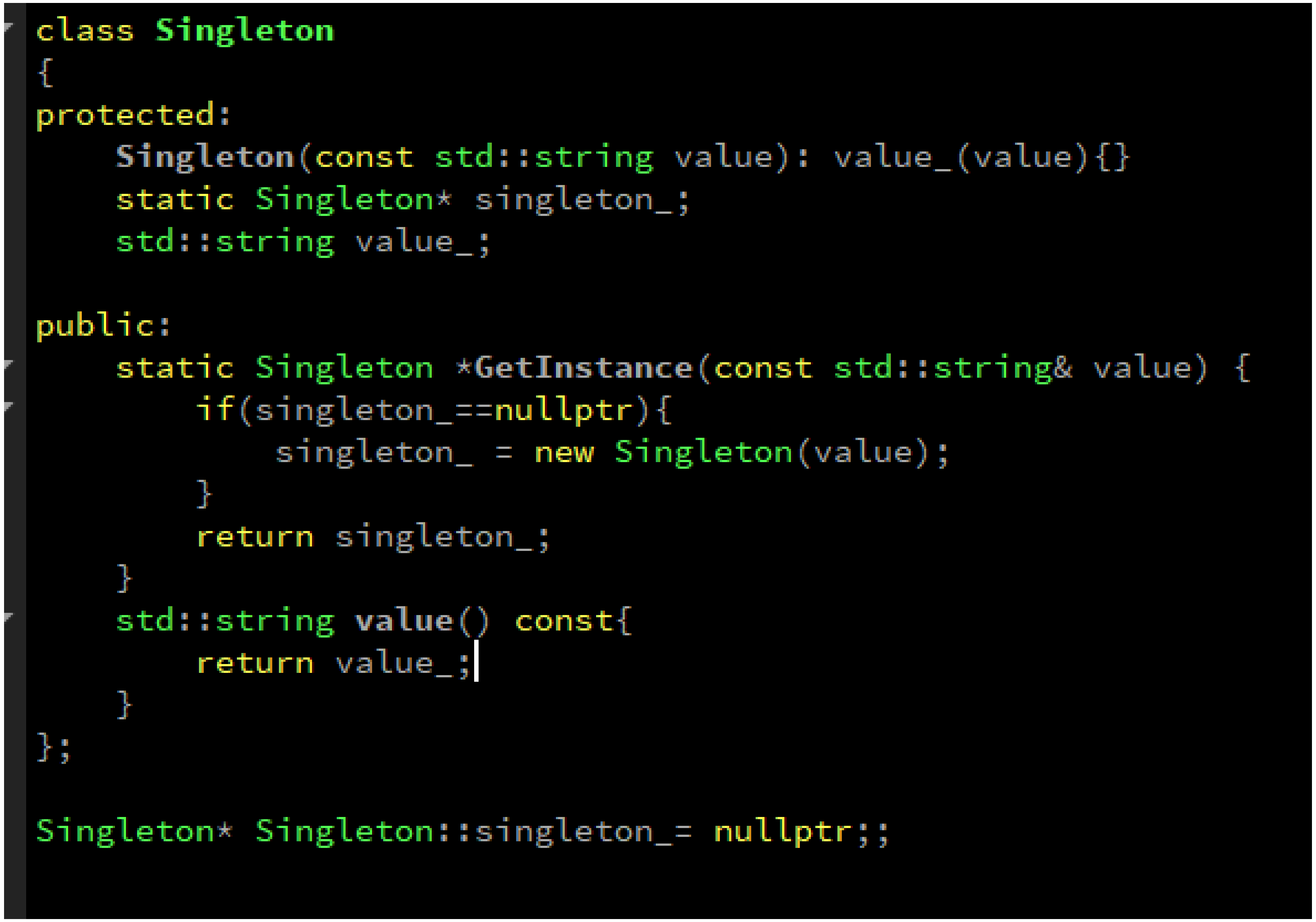
- **Concurrent Hash Table:** is built using the concurrent lists we developed

```
#define BUCKETS (101)

typedef struct __hash_t {
    list_t lists[BUCKETS];
} hash_t;

void Hash_Init(hash_t *H) {
    int i;
    for (i = 0; i < BUCKETS; i++)
        List_Init(&H->lists[i]);
}

int Hash_Insert(hash_t *H, int key) {
    return List_Insert(&H->lists[key % BUCKETS], key);
}

int Hash_Lookup(hash_t *H, int key) {
    return List_Lookup(&H->lists[key % BUCKETS], key);
}
```

Figure 29.10: **A Concurrent Hash Table**

VI- APPLICATION

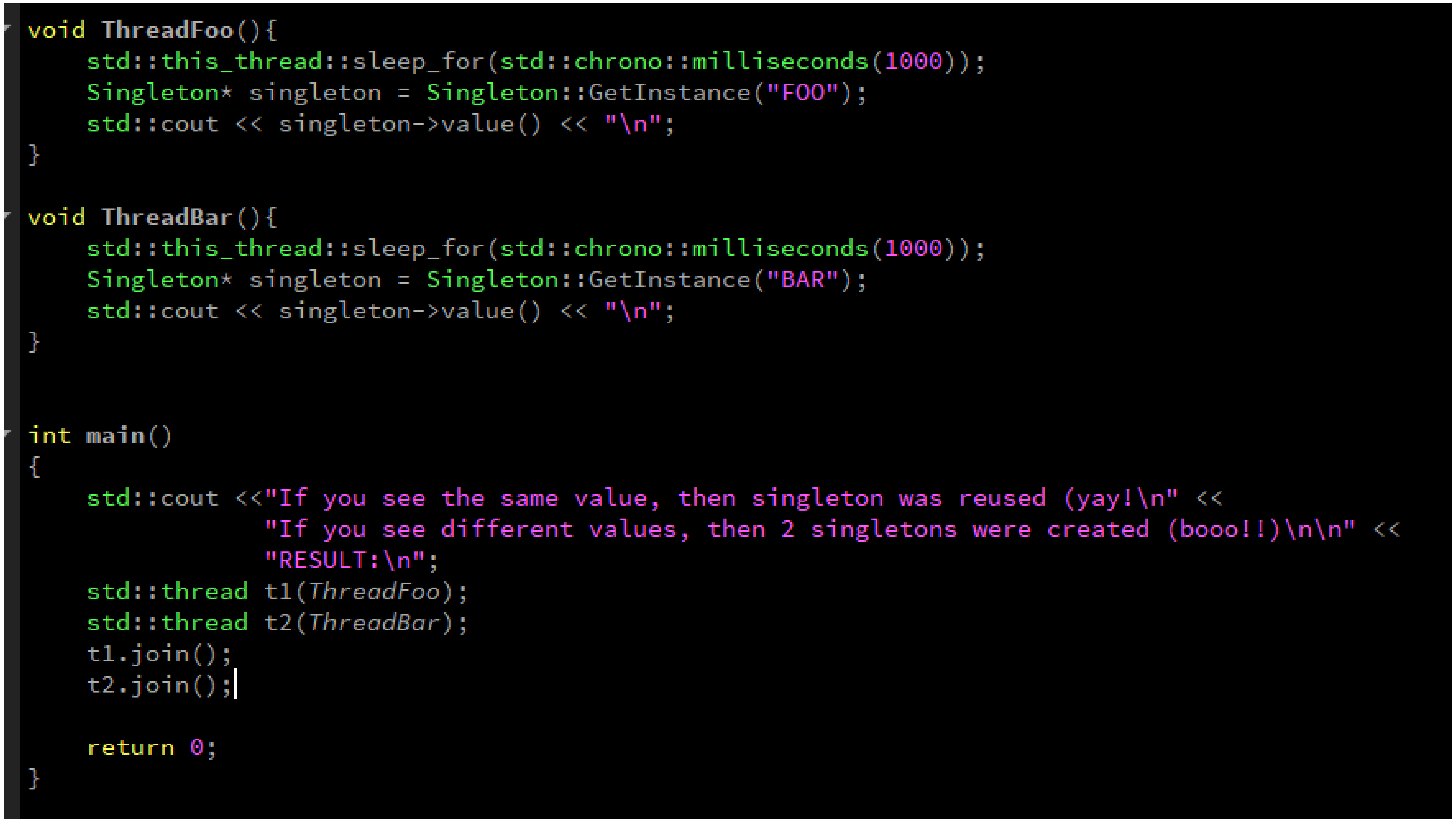
Singleton pattern



```
class Singleton
{
protected:
    Singleton(const std::string value): value_(value){}
    static Singleton* singleton_;
    std::string value_;

public:
    static Singleton *GetInstance(const std::string& value) {
        if(singleton_==nullptr){
            singleton_ = new Singleton(value);
        }
        return singleton_;
    }
    std::string value() const{
        return value_;
    }
};

Singleton* Singleton::singleton_= nullptr;;
```

VI- APPLICATION

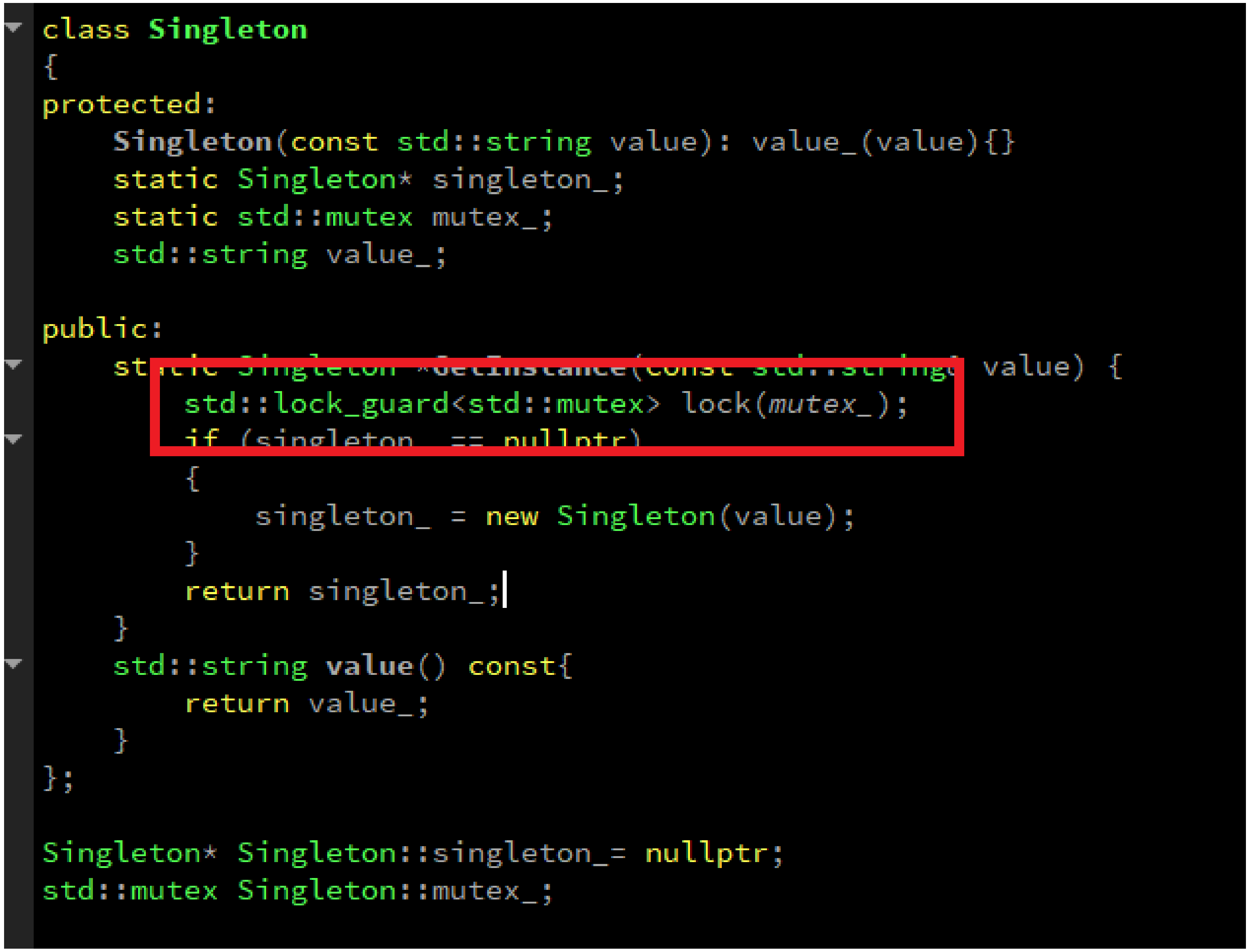
Issue



```cpp
void ThreadFoo(){
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));
    Singleton* singleton = Singleton::GetInstance("FOO");
    std::cout << singleton->value() << "\n";
}

void ThreadBar(){
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));
    Singleton* singleton = Singleton::GetInstance("BAR");
    std::cout << singleton->value() << "\n";
}


int main()
{
    std::cout <<"If you see the same value, then singleton was reused (yay!\n" <<
                "If you see different values, then 2 singletons were created (booo!!)\n\n" <<
                "RESULT:\n";
    std::thread t1(ThreadFoo);
    std::thread t2(ThreadBar);
    t1.join();
    t2.join();

    return 0;
}
```

VI- APPLICATION

Thread singleton



```
class Singleton
{
protected:
    Singleton(const std::string value): value_(value){}
    static Singleton* singleton_;
    static std::mutex mutex_;
    std::string value_;

public:
    static Singleton *GetInstance(const std::string& value) {
        std::lock_guard<std::mutex> lock(mutex_);
        if (singleton_ == nullptr)
        {
            singleton_ = new Singleton(value);
        }
        return singleton_;
    }
    std::string value() const{
        return value_;
    }
};

Singleton* Singleton::singleton_= nullptr;
std::mutex Singleton::mutex_;
```

KEY CONCURRENCY TERMS

These four terms are so central to concurrent code that we thought it worth while to call them out explicitly

- A **critical section** is a piece of code that accesses a shared resource, usually a variable or data structure.
- A **race condition** (or data race) arises if multiple threads of execution enter the critical section at roughly the same time; both attempt to update the shared data structure, leading to a surprising (and perhaps undesirable) outcome.
- An **indeterminate** program consists of one or more race conditions; the output of the program varies from run to run, depending on which threads ran when. The outcome is thus not deterministic, something we usually expect from computer systems.
- To avoid these problems, threads should use some kind of **mutual exclusion** primitives; doing so guarantees that only a single thread ever enters a critical section, thus avoiding races, and resulting in deterministic program outputs.

REFERENCES

- https://pages.cs.wisc.edu/~remzi/OSTEP/
- https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html
- https://www.geeksforgeeks.org/thread-functions-in-c-c/
- https://en.cppreference.com/w/cpp/thread/thread
- https://www.geeksforgeeks.org/multithreading-in-cpp/
- https://refactoring.guru/design-patterns/singleton/cpp/example

Thank you!